Department of Communications and Networking

Poor Man's Content Centric Networking (with TCP)

Pasi Sarolahti, Jörg Ott, Karthik Budigere, Colin Perkins

Aalto University | SCIENCE + TECHNOLOGY | WORKING PAPERS

Poor Man's Content-Centric Networking (with TCP)

Pasi Sarolahti, Jörg Ott
Karthik Budigere
Aalto University
Department of Communications and Networking
P.O. Box 13000, FI-00076 Aalto
Finland
{psarolah,kbudiger}@cc.hut.fi
jo@comnet.tkk.fi

Colin Perkins
University of Glasgow
School of Computing Science
Glasgow G12 8QQ
United Kingdom
csp@csperkins.org

ABSTRACT

A number of different architectures have been proposed in support of data-oriented or information-centric networking. Besides a similar visions, they share the need for designing a new networking architecture. We present an incrementally deployable approach to content-centric networking based upon TCP. Content-aware senders cooperate with probabilistically operating routers for scalable content delivery (to unmodified clients), effectively supporting opportunistic caching for time-shifted access as well as defacto synchronous multicast delivery. Our approach is application protocol-independent and provides support beyond HTTP caching or managed CDNs. We present our protocol design along with a Linux-based implementation and some initial feasibility checks.

1. INTRODUCTION

The way in which the Internet is used has evolved over time, moving from being a platform for providing shared access to resources, to become a means for sharing content between groups of like-minded users. Today, three forms of content sharing dominate the traffic in the Internet: web peer-to-peer applications, and media streaming (both via HTTP and other protocols), with the latter gaining in share [17].

This shift in network usage has driven two parallel strands of research and development. Network operators and content providers have responded with a host of pragmatic engineering solutions to provide content centric caching and media distribution within the existing network architecture, while the research community has studied a range of alternative architectures. Content distribution networks (CDNs) provide the pragmatic approach to data-oriented networking. They provide a transparent redirection and content caching infrastructure, with proactive content push, that can direct requests to the closest replica of content to improve access times and reduce network load. Distribution and caching of content in CDNs is commonly applied at the object (HTTP resource) level, and so is generally limited to this single application protocol. This works very well for web data, and modern adaptive HTTP streaming applications that segment media flows into short cachable chunks, but is problematic for live or open-ended content, and for non-HTTP traffic.

The pervasiveness and effectiveness of this CDN infrastructure exert a strong pull on applications to use HTTP-based content distribution. In some cases this may be appropriate, the ongoing migration of on-demand video streaming to use rate-adaptive chunked distribution over HTTP [1] is a timely example, but in other cases, that infrastructure is a poor fit. Large-scale live television distribution provides one example that does not fit well with the CDN model, and instead relies on IP multicast for effective, low-latency, transport. Peer-to-peer applications also do not benefit from these mechanisms, and instead construct their own overlays to provide good performance (some operators have started to deploy mechanisms to help overlays find suitable well-located peers (e.g., [24]) to enable caching and local content distribution). The result is that non-HTTP applications are disadvantaged: there is still no effective application neutral content distribution infrastructure.

In parallel to these engineering efforts, the research community has seen an upsurge of interest in data-oriented, and content- and information-centric networking (ICN), to re-architect the network to better suit the traffic mix and needs of all its users. Several architectures have been proposed as a future networking approach to address the above trends, and to allow identification and access of content inside the network, irrespective of the communication context. Three prominent examples include DONA [16] and PSIRP [15, 23], which operate at the packet level, as well as CCN [13] which uses larger *chunks* as basic unit of content distribution. These architectures, however, are clean slate designs that typically require intrusive modifications to the communication stack or even the entire network architecture. Moreover, naming and routing issues are not yet solved to scale.

In this paper, we propose a novel poor man's approach to content-centric networking. We use the idea of uniquely-labelled data units within content flows, coupled with packet-level caching of content at routers, to build a pervasive content caching and distribution network. This approach can provide much of the benefit of the proposed ICN architectures, but is incrementally deployable in today's Internet, and builds on current protocol stacks and standard transport protocols, such as TCP. In contrast to operating on HTTP-level objects, caching done at the packet level, independent of the higher layer protocols, offers more fine-grained caching and retrieval, and achieves broad applicability: to the web, to streaming, and to peer-to-peer overlays.

We implement packet-level caching on top of TCP flows, and preserve the traditional endpoint views and APIs. This includes the notion of using IP addresses and port numbers for connection identification by endpoints, client-server and peer-to-peer interaction

.

paradigms, and the use of URIs for resource-level content identification.[1] We add per-packet labels at the sender side to make packets identifiable as content units in *caching routers* inside the network; these labels are invisible to receivers. Caching routers operate opportunistically and may keep packets cached for some time after forwarding them. Caching routers keep state for any subset of flows with labeled packets, so that they are able to service requests across different flows autonomously from cached packets. This architecture supports two complementary uses of packet caching:[2]

- *Pseudo-multicast:* a synchronous delivery of data from each cache router to a virtually unbounded number of downstream cache routers and receivers where these are identified as being subscribed for the same data.
- *Packet-level caching:* time-shifted access to popular content from different receivers (as a function of the cache size and the request diversity).

This paper hints that we may gain the performance benefits of ICN relating to caching and delivering data to multiple receivers in a multicast fashion, while being able to keep the end-to-end transport state consistent. Our solution, *Multi-Receiver TCP (MRTCP)*, is based on TCP as the primary transport protocol for our target application class. It requires a modest amount of modifications to sending (usually server) TCP/IP stack, but it works with unmodified off-the-self TCP receivers (usually clients). Caching routers must be modified to be able to take advantage of MRTCP, but legacy routers and servers interoperate seamlessly with our enhancement.

Our main contributions are primarily of conceptual nature: 1) We present a complete design for a TCP enhancement that enables flow-independent content caching with legacy receivers (Section 3) and allows for improvements when receivers are MRTCP-aware (Section 4). 2) We outline how applications need to operate to make use of MRTCP features (Section 6). 3) We have implemented MRTCP in the Linux kernel, an MRTCP caching node, and of enhancements to an open source web server to support MRTCP using which we carry out initial interoperability tests with legacy clients (Section 7). We leave an encompassing performance evaluation for further study but rather focus on implementation feasibility in this paper.

Network operators and content providers have an incentive to deploy MRTCP since its pseudo-multicast and pervasive packet caching can significantly reduce their network load, albeit with increased router complexity. The incentive for receivers to deploy MRTCP is lower, unless those receivers participate in peer-to-peer overlays or host local content, but deployment to edge routers achieves much of the benefit for in-network state reduction (see Section 5).

We opt for implementation above the IP layer because we do not want to affect IP forwarding in non-caching routers and want to be independent of the IP version. Moreover, we would like to support generic caching independent of a particular application protocol and thus stay below the application layer. We also want to minimize the depth of extra packet inspection effort in caching routers; we believe that packet-level handling of TCP in caching routers might be within the reach in the near future (e.g., [5]). Section 9 discusses the trade-off of implementation at other layers.

[1]Despite the fragility of URIs caused by their inclusion of content location and access method, in addition to content identification, users have accepted them as a viable and familiar way to convey pointers to content.

[2]As a side effect, we could also support packet retransmissions within the context of a TCP connection, but this is not in our present focus.

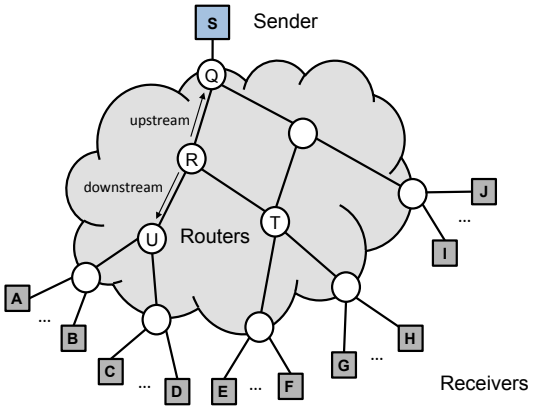

Figure 1: Sample network topology with one sender S, a number of receivers A, ..., J, and routers Q, R, T, U, ...

2. OVERVIEW AND RATIONALE

In today's Internet, content identification is rather coarse-grained, with objects, or chunks thereof, being identified by URIs, or other application protocol specific identifiers. These identifiers generally conflate a name for the content with an indirection mechanism that can be used to locate the content, for example using the DNS, or a distributed hash table. Content delivery protocols typically specify a name for the content to be retrieved at the start of a transaction, but do not usually identify the individual data packets as part of the content being sought. Content retrieval occurs subsequently as a sequence of packets that are not self-identifying, and are linked to the content identification only by the distribution protocol. The link between a content identifier and a packet flow is broken with a new resource is identified (usually by a new retrieval request), or when the connection is closed.

The link between named content and packet flows is further weakened by protocols such as HTTP and RTSP that support content type negotiation when requesting a piece of named content. Under such protocols, receivers may, for example, specify preferred encodings or indicate their user agent type, and senders may provide a suitable variant of a resource in return. This means that the packets delivered in response to a request for a particular named content item may differ depending on who requested that content.

In contrast to this present-day behavior, a key requirement in our design of poor man's content-centric networking is being able to identify packets comprising some content inside one TCP connection, and make those packets re-usable across TCP connections. To achieve this, we split up the interaction between sender and receiver into phases—*content identification* and *content retrieval*—switching between which is controlled by the sender-side application.

Consider the example network shown in Figure 1, where receivers A through J fetch content originating from some original sender S. The network comprises a set of routers, of which those labeled Q, R, T, and U are upgraded *caching routers* we discuss below, while the others are standard IP routers. Content is disseminated in a downstream direction, while requests and acknowledgments flow upstream. The basic operation of MRTCP is depicted in Figure 2(a) using HTTP as an example. Initially, the sender and a

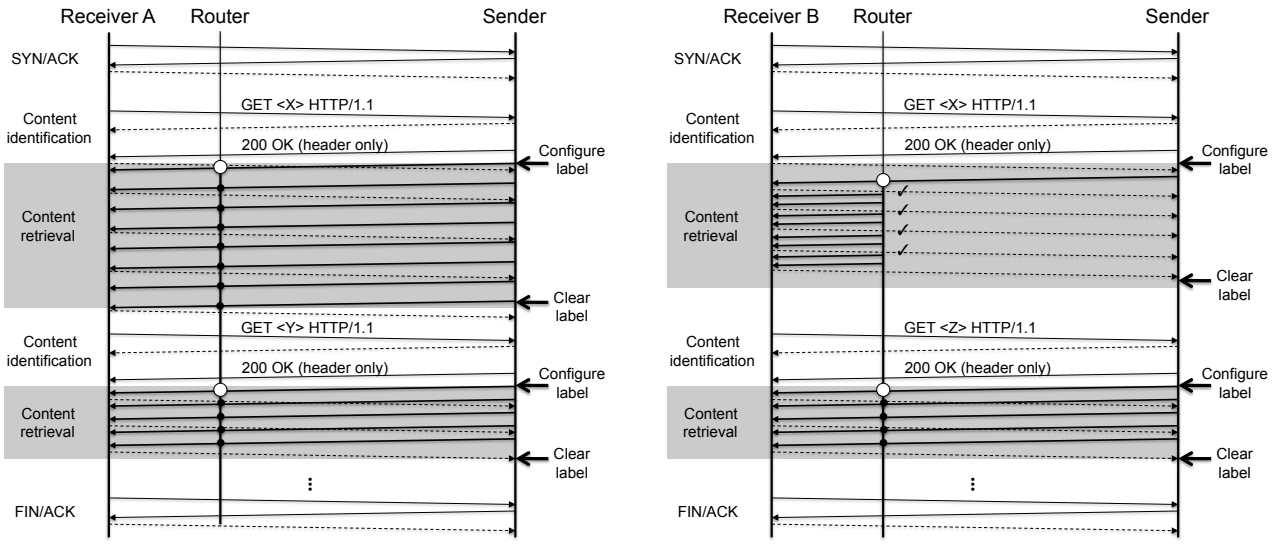

Figure 2: (a) (left) Basic MRTCP interaction between a receiver (e.g., a web browser) and a sender (e.g., a web server). The gray-shaded areas show content retrieval with labeled and thus cachable packets (thicker arrows) whereas the the white areas show non-cachable packets. Caching routers only recognize labeled packets (which they may cache, indicated by black circles) and ACKs for flows containing such; they ignore all others. (b) (right) After the first data packet has primed a router about the content unit retrieved via a flow (white circle), the router may respond with cached packets and mark this in the forwarded ACKs. All control packets (SYN, FIN, ACK) travel end-to-end.

receiver are in the content identification phase (white background) and exchange an upstream request to specify the content to retrieve, and a corresponding downstream acknowledgment: the HTTP GET request and the 200 OK header response in this example. The data packets sent during this phase are unlabeled and so considered non-cachable so that routers Q, R, T and U do not look at them.

Based upon the negotiated piece of content, the sender decides how to label the packets carrying that content, and indicates this via the Sockets API to the underlying transport ("configure label"). If needed, multiple labels may be used for a single resource and labels updated during the retrieval, e.g., if the resource is large (see below). Labeling packets signifies the beginning of the content retrieval phase (shaded gray in Figure 2(a)). During this phase, only labeled data packets, that may be cached by the upgraded routers Q, R, T and U on the path, are transmitted. The connection returns to the content identification phase when the label is cleared ("clear label"), and is then ready for identification of the next piece of content. The content identification and content retrieval phases may be alternated repeatedly until the connection is torn down. Note that the two directions of a connection are treated independently as it is up to the respective sender to indicate when and how to label data packets.

There are two variants of MRTCP: (i) a stateful variant, that requires a small amount of state at the routers for those flows they decide to cache, but works with standard TCP clients; and (ii) a stateless variant, that does not require per-flow state at the routers, but requires modifications to TCP clients. Both variants use the same labeling mechanisms to identify content in packets, and a network can support both variants at the same time, allowing gradual deployment of MRTCP. Also, the same MRTCP sender implementation works with both variants as MRTCP senders can indicate their support during the SYN/ACK handshake. In this paper, we focus primarily on the stateful variant that works with legacy clients, but we also outline the stateless variant in section 4.

MRTCP uses normal end-to-end transport connections and the end-to-end TCP signaling works (nearly) as before: both endpoints perform the SYN/ACK and FIN/ACK handshakes and the sender receives all acknowledgements. Thus, the sender becomes aware of all interactions, can engage into content identification and negotiation, and is subsequently able to trace the transmission progress.

The sender labels packets according to their contents as specified by the application. This is done via a small extension to the Sockets API. The label is then included in-band with the data packets. A TCP connection may arbitrarily mix labeled and unlabeled packets and different labels may be used as well. It is up to the sender to ensure that packet boundaries match content labels, and are reusable across connections. For example, in an HTTP response, the start-line and headers are sent separately in the first packet(s) so that the content starts on a packet boundary (ensuring framing is maintained is a sender-side modification to TCP, but needs no protocol or receiver-side changes). Routers observe passing packets and may cache (any subset of) the labeled ones using the label as the key to the cached content. In Figure 2, this is indicated by the white and the black circles.

Data packets lead to acknowledgments sent by the receiver. For flows that contained labeled packets before—this recognition and flow-to-label mapping is indicated by the white circles in the figure—the ACKs are noticed by routers and trigger (re)transmissions of packets from the router cache if fitting data is available. If so, this reduces the data transmission times and the amount of traffic at the sender end.

Transport-level ACKs are exchanged end-to-end, but MRTCP-aware routers will add a TCP option to the acknowledgments when they are processed. The option is used to coordinate the transmis-

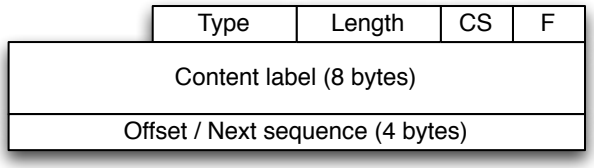

Figure 3: MRTCP_DATA / MRTCP_ACK option.

sion of packets, and it tells which are the next data packets to be sent, and how many packets are allowed to be transmitted in response to the acknowledgment. (see figure 2(b) for retrieval of the resource). This way the repeated transmission by other (upstream) routers and the sender can be avoided. The sender is ultimately responsible for providing reliability, and will transmit packets not cached by any router, and retransmit those packets lost end-to-end.

The sender application does not notice the difference between packets being sent by intermediary routers or by the sender system: it feeds the full content into the MRTCP connection via the Sockets API. It is the sender side kernel MRTCP implementation that decides per connection which of these packets are actually sent and which can be safely discarded. As a consequence, the MRTCP approach is suitable for reducing data transmission, but does not help local data access optimization.

In the following section, we describe the protocol details of the stateful variant of MRTCP, built on TCP because of its universal deployment (MRTCP is conceptually easier to implement with SCTP than with TCP, since SCTP natively preserves frame boundaries, but the lack of SCTP deployment makes it a less attractive base for development).

3. MRTCP: CONTENT-AWARE TCP

MRTCP extends TCP to become content-aware on a per-packet basis. As discussed in Section 2, an MRTCP connection is subdivided into phases of labeled and unlabeled packets. Packets with the same label carry pieces of the same content, but multiple labels may be used for larger content items. The labeling allows MRTCP-enabled caching routers to identify resources and store related packets and, for the stateful case we discuss in this section, to determine which resource a flow is carrying at a given point in time so that they can respond to ACKs with the right data packets.

3.1 Content Identification: TCP Options

Labels are set by the sender and are carried as a TCP option so that they are ignored by legacy TCP receivers. MRTCP uses the *MRTCP_DATA* option to indicate to which content item a particular data packet belongs. The option, as illustrated in Figure 3, contains a content label (*h_label*) that identifies the resources, and a content offset (*h_offset*). In this work we assume that the content label is 8 bytes, although there are benefits in choosing a larger label, weighing against the cost of using the scarce TCP option space, as will be discussed later. The content offset is 4 bytes, and indicates the relative offset of the TCP payload in bytes from beginning of the content item. There is also a 4-bit space for flags (F), and a 4-bit *h_cansend* field (CS) that is used in MRTCP acknowledgments.

For now it is sufficient to assume that the content label is any arbitrary byte string identifying the content item, chosen such that the collision of content labels for two separate objects is unlikely. For example, the label could be a self-certifying identifier that binds the label to the content in some way. We will discuss the security aspects of content labeling later in this paper.

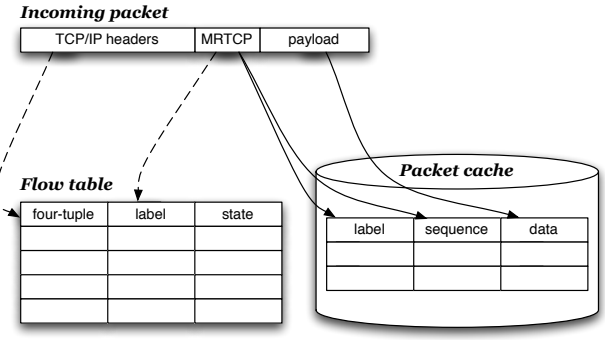

Figure 4: Data structures in flow-aware MRTCP router.

The relative content offset allows different receivers to request a particular segment of content, in spite of randomly initialized TCP sequence numbers in their respective TCP connections. With a four-byte content sequence number, if a resource is larger than 4GB, multiple labels have to be used. Also smaller resources may use multiple labels, as will be discussed later.

Content is acknowledged using the *MRTCP_ACK* option. The structure of the option is similar to MRTCP_DATA, but this option informs upstream routers about the next content sequence that should be transmitted (*h_nextseq*), either by an intermediate cache, or by the sender. MRTCP_ACK also makes use of the *h_cansend* field, by which a downstream client or router can tell how many packets the upstream nodes are allowed to send.

3.2 Receiver operation

For the stateful variant of MRTCP routers, we assume legacy TCP receivers that we are not able to influence. What is important from an MRTCP perspective is that they ignore unknown TCP options and just process the data contained in the packet[3]. The legacy receivers carry out their normal operation, and may implement different flavors of TCP: they may acknowledge every or every other packet, continuously update the receive window, possibly applying window scaling, and use different TCP options, such as timestamps or selective acknowledgments. However, the MRTCP sender may choose to disable some of these options during the initial negotiation, to save in the precious TCP option space also needed for MRTCP to operate.

TCP receiver implementations may be sensitive to significant packet reordering and may perform other (undocumented) sanity checks before accepting data. Therefore, MRTCP is designed to maintain ordered segment transmission as much as possible.

3.3 Router operation with legacy receivers

A *flow-aware MRTCP router* contains two tables, as shown in Figure 4: the *flow table* and the *packet cache*. The flow table is used to track TCP connections that carry labeled content in cache or to be cached. Each connection is identified by the source/destination IP address/port 4-tuple. The table stores the content label (*mr_label*) that is currently transmitted across this TCP connection along with the transmission progress of the resource, and some TCP state variables necessary for feeding packets into the connection:

- *Sequence offset (mr_offset)*, The TCP sequence number corresponding to the first byte of the content item that is cur-

[3]According to measurements made few years ago, unknown TCP options are correctly ignored by nearly all Internet hosts, and will not hamper the normal TCP communication [19].

rently in transit. This is used to match sequence numbers from incoming acknowledgments to content sequence numbers that are relative to the beginning of the content, for cache matching.

- *Next content sequence number (mr_nextseq)* to be sent. This field is updated based on the number of bytes in data segments received from the sender, and bytes acknowledged in MRTCP_ACKs arriving from the receiver. If a router gets a larger sequence number from either side than what it has stored in *mr_nextseq*, it sets *mr_nextseq* to this sequence number. The router uses this parameter to choose the next cached segment to be sent. After sending cached data in response to an ACK, the router increases the value of this field by the number of bytes sent, and reports the new value to upstream nodes in the *h_nextseq* field of an MRTCP_ACK option.

- *Last acknowledgment number (mr_lastack)* received. Acknowledgements are for the next content byte that is expected. The acknowledgement number is used to distinguish duplicate acknowledgments from those that advance the window. The router does not send cached content on duplicate acknowledgements, to avoid the possibility of (re)transmitting duplicate packets to the network. Letting MRTCP routers retransmit lost content from their cache is an interesting optimization, however, which we discuss more in Section 8.

- *Congestion window (mr_cwnd)* maintains the number of bytes allowed to be in transit, as indicated by the difference of *mr_nextseq* and *mr_lastack*. There is also a duplicate ACK counter (*mr_dupacks*) for identifying three consecutive duplicate acknowledgments as an indication of congestion.

The *packet cache* stores named content items. The tuple of *(content label, content offset)* identifies the cached data, which corresponds to the payload of the TCP segment (data, length) that contains the content. Inside the cache, packets are organized according to their labels so that all-or-nothing decisions can be taken for caching and evicting packets.

Figure 5 outlines the operation of an MRTCP router. When such a router receives a data packet with MRTCP_DATA TCP option, it caches the payload data based on the content label (h_label) and offset (*h_offset*) extracted from the packet header. If this is the first packet with the given content label in a particular TCP flow, the router stores the 4-tuple identifying the flow along with the current content label in the flow table (figure 5(a)). The router also stores the packet's TCP sequence number to be able to determine the content offset in subsequent TCP segments and acknowledgments. The router initializes the next content sequence field to the content sequence that would follow the current segment, and the last content acknowledgment field to 0. If the content label changes, i.e., transmission of a new content object starts, the content label for the flow is set accordingly, and the sequence number and acknowledgment fields are re-initialized. The router also updates the state information in the flow table according to the TCP and MRTCP headers (figure 5(b)).

When an acknowledgment from the client comes in, the router checks if information about the connection is available in the flow table (figure 5(c)). If there is no matching 4-tuple, the packet is forwarded without any further action.

If the 4-tuple matches, but the incoming acknowledgment does *not* include an MRTCP_ACK TCP option, the router performs the following operations:

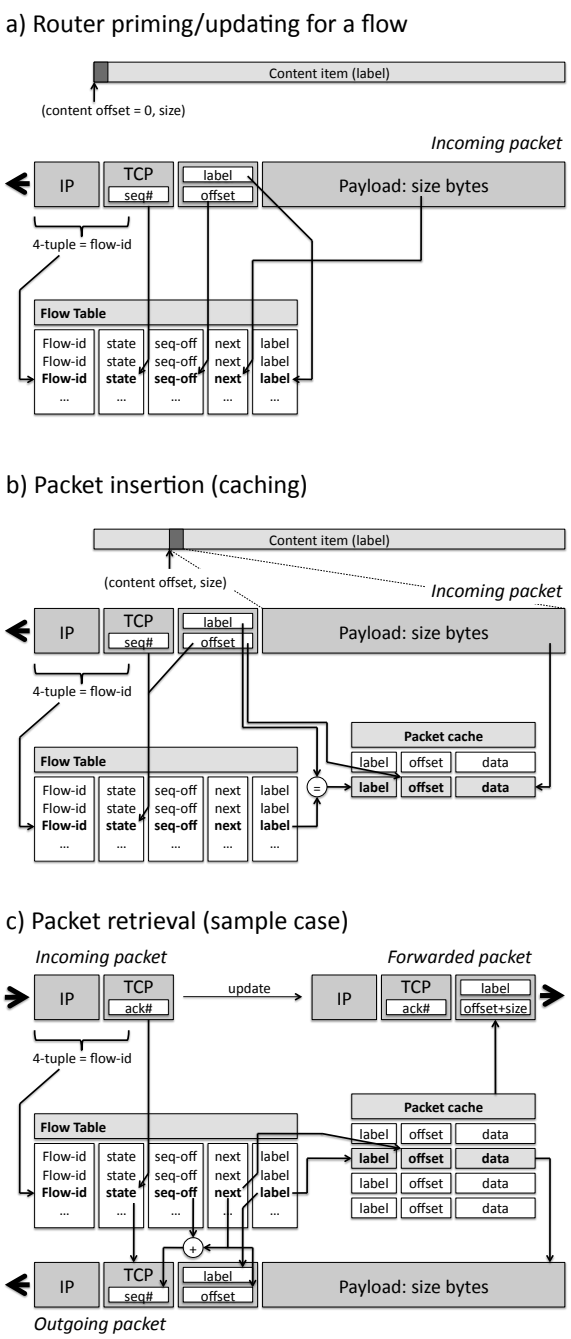

Figure 5: Overview of caching-related operations in a flow-aware MRTCP router.

1. The router updates *mr_lastack* based on the acknowledgment number in the packet. It calculates new congestion window by procedures discussed in Section 3.5.

2. The router checks if the congestion window and the receive window allow data to be transmitted (not shown in the figure). It determines the amount of outstanding data using the information in *mr_nextseq* and *mr_lastack*, and compares this to a locally maintained congestion window to determine

how many new packets can be transmitted (local variable *"can_send"*). For these calculations the router can assume an approximate fixed packet size, for example based on the largest segment size observed in the flow. The router also needs to check that it does not exceed the receiver's advertised window when it calculates the *"can_send"* value. When determining the *can_send* value, the router should ensure that it does not request transmission of packets that may have been triggered by earlier acknowledgments at some of the upstream routers. For example, if an earlier acknowledgment had allowed transmission of two new packets, and *mr_nextseq* is advanced just by one packet, the router should set the *can_send* value to 0, because the second packet might be on its way to the router.

3. If *can_send* is positive, the router uses *mr_label* and *mr_nextseq* to determine if it has the next consecutive packet in its packet cache. If can_send is 0, or the next packet cannot be found in the cache, the router does not send any cached data, and forwards the received acknowledgment towards the TCP sender after adding an MRTCP_ACK option. This MRTCP_ACK option uses *mr_label* for the content label in the option, the (possibly updated) *mr_nextseq* for the next sequence field, and the final value of *can_send* for *h_cansend*.

4. If the next packet was found in the cache, and the value of *can_send* allows sending it, the router builds a new packet, using the information in the recent acknowledgment and in the flow table to build the TCP and IP headers, with the MRTCP_DATA option to assist the downstream routers. The router uses the IP address and port of the original sender as the source address. The router then decrements *can_send* by one and updates *mr_nextseq* based on the highest sequence number transmitted (which should be larger than the previous value of *mr_nextseq*). This procedure is then repeated from step 3, until *can_send* reduces to 0, or the next packet cannot be found in the cache.

When an MRTCP router receives an acknowledgment that was previously processed by a downstream MRTCP router, and therefore has the MRTCP_ACK option, it follows the above procedure with the following small exceptions: 1) before evaluating what data to send, the router updates the *mr_nextseq* field based on the information from the arriving MRTCP_ACK option, if the incoming value is larger than currently stored; and 2) instead of maintaining an own congestion window, it uses the *h_cansend* parameter from the MRTCP_ACK option together with the receive window to determine how many (if any) further packets can be sent from the local cache. This leads to conservative behavior: the local transmission rate never exceeds a smaller congestion window on a downstream router. When an incoming acknowledgement contains MRTCP_ACK option, the router can only send data from the point indicated by the *h_nextseq* field in the option, in other words, the router that first inserted the MRTCP_ACK option has the control over which packets are sent. Otherwise multiple stateful MRTCP router might be sending same packets in parallel. If a router sends data packets in response to a packet with MRTCP_ACK option, it reduces the *h_cansend* field in the option by the number of packets sent, and updates the *h_nextseq* field accordingly, before passing the acknowledgment towards the TCP sender.

Building the TCP and IP headers based on an incoming acknowledgment and the information in flow table is fairly straight-forward. Choosing the value for receiver's advertised window is a bit more problematic because the router cannot know the buffer state at the end host. Since flow state gets established by a labeled data packet, the router remembers the last value seen from the sender and uses it when creating data packets.

If an MRTCP_ACK option is present in a packet, a router knows that there is another downstream router that is MRTCP-aware and is tracking the flow state for the TCP connection. This allows for the router to drop its own flow state, and send cached packets just based on information in incoming acknowledgments. This requires that a valid TCP sequence number is also carried with the acknowledgments, so that the router is able to compose a valid TCP segment that hits the expected receive window. We will discuss more about stateless MRTCP router operation in Section 4.

3.4 Sender operation

An MRTCP sender initiates a TCP connection as normal, except that it may refuse some TCP options (for example, TCP timestamps) during the SYN/ACK handshake to ensure enough option space for the MRTCP extensions. The TCP maximum segment size is also negotiated such that there is enough room for the content labels.

Within a TCP connection, the MRTCP sender operates as normal TCP until an application indicates the start of a labeled content by assigning a label to the outgoing data. From this point, on the sender includes MRTCP_DATA options in packets using the content label set by the application. Like MRTCP routers, an MRTCP sender maintains a mapping between the content offset and the TCP sequence number in the socket control block. The application can change the content label within the same TCP connection at any time (for example, when starting a new object in the case of persistent HTTP connections or when the size exceeds 4GB), and it can indicate the end of a content item, after which the sender stops including the MRTCP_DATA options in the packets. Setting and resetting the label is done via a local socket operation (see below).

When a sender receives an acknowledgment, it adjusts its transmission window normally based on the information in the acknowledgment. If there have been MRTCP-aware routers on the connection path, the sender will receive MRTCP_ACK options with the acknowledgments. In this case, instead of its own congestion window, the sender will use the *h_cansend* field in determining whether it can send data in response to the acknowledgment, and the *h_nextseq* field in determining what packet to send next. If the value of *h_cansend* is 0, it is likely that routers along the path have sent packets in response to the acknowledgment, and the sender does not send any data. The sending application is responsible for ensuring that labelled (and potentially cachable) content is identical on all (re)transmissions, whether sent or not.

When an MRTCP_ACK option arrives, the TCP sender checks if the value of *h_nextseq* is larger than the current SND.NXT, and in such case updates SND.NXT based on the incoming value. This way the sender avoids re-sending data that has been transmitted by one of the routers. The sender does not increase its congestion window, because it cannot take a reliable sample of the network conditions from the acknowledgment. Likewise, it is not possible to take round-trip time samples. However, from the incoming acknowledgment the server gets information about the progress of the transport, and can move along its transmission window accordingly.

The sender is responsible for sending content to fill in gaps if some packets are not cached in routers, and for performing retransmissions to repair end-to-end packet losses. If an incoming acknowledgment is a duplicate acknowledgment, the sender can trigger fast retransmission and the following loss recovery normally. If

```
for each incoming acknowledgment (pkt):
  if (pkt.ack == mr_lastack) {
     mr_dupack++;

     if (mr_dupack == 3)
         mr_cwnd >> 1;

     can_send = 0;
  } else if (pkt.ack > mr_lastack) {
     mr_cwnd++;
     mr_dupack = 0;
     mr_lastack = pkt.ack;

     rack = pkt.ack - mr_offset;
     can_send = cwnd - (mr_nextseq - rack);
  }
```

Figure 6: Congestion control algorithm at MRTCP router.

the retransmissions are sent containing an MRTCP_DATA option, the retransmitted data can be cached. Our Linux implementation allows this, because data is packetized when it is received from the socket, and the content label and offset information is attached to at that point. If sender receives acknowledgment without the MRTCP_ACK option, it reacts to it normally, using its local state, e.g., for congestion window and next sequence number to be sent.

3.5 Flow and Congestion Control

An MRTCP router needs to manage its transmission rate to avoid overloading the network, and to keep within the advertised receiver window. Therefore, an MRTCP sender and stateful routers need to manage congestion window similar to any normal TCP sender.

At an MRTCP router it is desirable to keep the amount of per-flow state and per-packet processing to a minimum. Furthermore, we do not want to maintain separate per-flow timers at the router. Therefore, we apply a simplified form of congestion control that roughly follows TCP's behavior with minimal amount of processing and state. The algorithm is shown in Figure 6.

In this algorithm, *pkt.ack* refers to the acknowledgment field of the incoming acknowledgment, and the *mr_** variables are as described earlier. The calculated received acknowledgement, *rack*, indicates the content sequence that was acknowledged, and *can_send* tells how many packets can be sent. The congestion window *mr_cwnd* is managed on per packet basis here, for the reasons of efficiency. The first half of the algorithm processes duplicate acknowledgments: in this case no new packets are sent by the router, and in case of three consecutive dupacks, the congestion window is halved. The second half of the algorithm processes an acknowledgment for new data, and increases the congestion window similarly to slow start. The algorithm then determines how many new packets can be sent. Commonly, depending on whether the receiver has applied delayed acknowledgments or not, the router will send two or three packets out for an incoming acknowledgment, if it can find the data in its cache.

If an acknowledgment contains an MRTCP_ACK option, the value of *h_cansend* field is also considered in addition to the above algorithm, and the smaller of the two values is used. Downstream routers can use MRTCP_ACK to indicate that they have used all of their congestion window quota, and signal to the upstream hosts that they should not create new packets in response to this acknowledgment.

The above algorithm does not fully compare to the standard TCP congestion control algorithm [2], and in fact seems more aggressive since it only applies slow start, without any retransmission timers. On the other hand, use of MRTCP caches has potential to significantly reduce the traffic from the upstream network path, so its overall effect is more likely to reduce congestion from network than to increase it.

As for flow control, an MRTCP router may misinterpret TCP's receiver window size field ($rwin$) in a TCP acknowledgment packet because the two ends may have negotiated window scaling. Because window scaling negotiation happens during TCP connection establishment handshake when content label is not yet announced, it is hard for an MRTCP router to detect a scaled window, without tracking all TCP SYN/ACK handshakes. As a rough approximation, the router checks if $rwin > 0$ and will send segments only in this case. We assume that a receiver will not report "silly windows" so that at least one segment transmission will be possible. The router then sends packets as allow by *can_send*. All packets sent by a router will be marked in the corresponding ACK traveling upstream (and recorded at every MRTCP router) so that the same packets will never be sent by another router (unless the ACK is lost); the recovery will take place at the MRTCP sender that will interpret the $rwin$ field correctly.

3.6 Operational limitations

The design of MRTCP is guided by the idea of simplicity. A router router does not need to implement complex algorithms and the per-flow state that is maintained can be inferred from any passing data packet, so that route changes or reboots are not an issue, and the router can safely discard flow state and is able to recreate it later if needed.

MRTCP-enabled routers avoid sending out-of-order segments. Therefore, if there is a packet missing in the cache, subsequent data is not sent before the missing packet arrives from an upstream router or from the original sender, potentially at the cost of increased round-trip time. While this may impede performance, not sending out-of-order segments avoids triggering duplicate ACKs at the receiver and, possibly, consequent retransmissions and congestion window updates at the sender. Moreover, the semantics of the *h_nextseq* field in the MRTCP_ACK is also problematic with out-of-order segments: An additional bit mask would be needed to indicate which packets were sent.

4. STATELESS MRTCP

Per-flow state in MRTCP routers is required to allow them to construct TCP packets representing cached content that fit properly into the sequence number space of the end-to-end TCP connection, and to manage an appropriate transmission rate. Because maintaining per-flow state may be expensive for busier routers, we now discuss how to use MRTCP in such way that a router does not need to keep any per-flow state, i.e., there is no flow table in the router. In this case, the corresponding state needs to be managed by one of the downstream routers, or the TCP receiver, which then needs to be modified to be able to support MRTCP. If the receiver supports MRTCP, it can provide the necessary mapping information in each ACK and the flow state in the routers is no longer needed. In the following we focus on the case where the TCP receiver is modified to support MRTCP.

We warn early on, that the stateless operation at routers is vulnerable to a wider range of security issues than when the routers manage per-flow state. In particular, it becomes easier for outside sources to pollute caches with arbitrary labels. Per-flow state at routers provides some tools to protect against such attack. We will return to this issue in Section 8.

MRTCP receivers use the same TCP options as does the stateful variant of MRTCP for acknowledgment, and are compatible with

any upstream stateful MRTCP routers on the path to the sender. The difference is, that the MRTCP_ACK option is added already at the receiver, A stateless router that receives the MRTCP_ACK option takes it as a request to send the next packet(s), that is indicated by the *h_nextseq* field. The maximum number of packets that the router can send is given in *h_cansend* field. Since the label and the offset are explicitly specified by the MRTCP receiver, indexing into the cache works straight away and no additional flow table mapping is required by the MRTCP router.

In addition, for a stateless router to work without the flow table, another option called TCP_STATE must be included by the receiver. This option contains the TCP sequence number to be included in the data packet for the first byte of the content offset sent by routers. The rest of the header fields for sending a cached content can be built from the TCP/IP headers of the incoming MRTCP_ACK packet: the source and destination IP addresses and ports are just reversed from the ACK packet. The concerns on selecting the receiver's advertised window discussed in the previous section apply also here: the router needs to choose a value that allows bidirectional traffic without overloading the receiving buffers too often. Moreover, MRTCP receivers accept TCP segments that have the ACK bit cleared and the acknowledgment sequence number set to 0, so that the routers do not need to track the sender-side acknowledgments either. In effect, the receiving TCP performs the state tracking on behalf of the routers so that these can be stateless.

Because stateless routers cannot carry out congestion control, MRTCP receivers need to maintain an equivalent of the congestion window (for example, using the algorithm in Section 3.5), and communicate the maximum number of packets that can be sent to the upstream routers. This information is sent in the *h_cansend* field of the MRTCP_ACK option. Routers will process this field similarly as in the stateful case, reducing it by the number of packets sent in response to the acknowledgment. Exploring different receiver-based congestion control mechanisms (such as [11]) is for further study.

The MRTCP router operation is mostly unchanged. The router uses the *h_cansend* field to determine how many packets to send, and the information from incoming packets and the TCP_STATE option to be able to build and send a valid TCP segment. Before forwarding the acknowledgment, the router reduces the *h_cansend* value in the option by the number of packets it sent in response to the ACK.

When a receiver supports MRTCP and manages the related connection state, it is unnecessary for a stateful router to include the state information in its flow table. Another possible optimization could be for the routers to include a "sent_by_router" flag in the MRTCP_DATA option when a packet is sent from the cache. The receiver could use this information for its congestion control heuristics, gaining the knowledge that the packet did not come from the original source. This could also tell a receiver to ignore the advertised window inside the packet but stick to the value last received from the source instead. We will investigate these ideas further as part of our future work.

5. CACHING CONSIDERATIONS

The basic operation of our MRTCP router is inspired by the router design considerations for ICNs as in [5]: packets are received and queued for forwarding, but data packets carrying a content label are at the same time indexed, stored, and kept accessible for later until they are overwritten by future packets. The total buffer may be larger than the share used for queuing. To simplify the organization of the cache, fixed sized memory cells can be used for storing packets.

As our cachable packets carry labels for identification, indexing them could use, e.g., a fixed set of bits from the label (or some hash function) and the content sequence number and utilize a hierarchy of fast and slower memory, as suggested in [5]. The indexing just needs to ensure that consecutive packets can be easily identified; a simple implementation might, e.g., allow storing up to 2^k consecutive packets by using the lower k bits of the sequence number.

For retrieving packets, if an MRTCP_ACK option is included, the label and next sequence fields are used in the same fashion (plus a full label comparison to check if it is the right packet). Otherwise, a more expensive indirect lookup is required via the flow state to obtain the label and the offset field. This indirection is needed only *at the first router with a matching flow* that then adds the ACK label. Since routers closer to the receivers usually see fewer flows and packets, the identification may be left to those routers closer to the edge, simplifying the task for more heavily loaded routers further in the core; the latter could simply ignore unlabeled packets passing through. We note that a network operator may deploy edge routers that support MRTCP, and gain most of the benefit they would if edge hosts were upgraded; this clearly eases deployment, since the incentive to deploy is on the network operator that benefits from caching.

As MRTCP routers cache opportunistically (unlike, e.g., [3]), they do not require coordination with any upstream or downstream routers, and may operate independently. This may also be the case for routers in a content-centric networking environment [5], but those expect suitably designed transport or application protocols. In contrast, we have to consider the interaction with TCP endpoints. For this reason, our protocol design tries to avoid out-of-order delivery of TCP segments and allows a router only to send consecutive segments in response to an ACK. Any gap needs to be filled in by an upstream router or the sender and causes extra delay. Consequently, this should be reflected in the router's caching strategy.

MRTCP routers manage content items, TCP flows, and cachable packets carrying pieces of content items as part of flows. Flow state is essential for responding to ACK packets but routers can only maintain state about a finite number of flows. When a new flow is detected and the flow table is full, we choose the flow with a probability of p_c for caching. Since p_c is evaluated upon each incoming packet, choosing p_c sufficiently small will statistically ensure that short flows don't get cached easily. If a flow passes this check, we select the least recently used flow as the one to be replaced.

Packets are considered with respect to the content items to which they belong, and independent of the individual flows referencing them. The basic idea is to cache packets that belong together, and to give precedence to packets for which content items already exist and are in active use. Ideally, all packets belonging to a resource should be stored and purged jointly when no longer requested, effectively implementing a least-frequently-used (LFU) or least-recently-used (LRU) policy on content items as a whole.

However, storing large content items as a whole may not be feasible and contradicts the basic idea of fine-grained caching using packets. Therefore, we suggest taking caching and purging decisions jointly on series of, e.g., 2^k, packets belonging to a resource. The packets, or packet series, per content item can additionally be managed using LRU for purging. In case a packet needs to be discarded, the oldest ones are dropped. If the size of a series falls below a certain threshold, the entire series is dropped.

This organization of per-item caching serves the two different target uses of MRTCP without the router being consciously aware. For the retrieval of cachable web objects, files, or on-demand stream-

ing, all packets of a content item will be accessed repeatedly as long as the resource is popular. Hence, packets from all areas of the resource (possibly even all its packets) are likely to remain cached. This also serves packet retransmissions well. For multicast-style synchronous streaming, only a relatively small window of packets (say, a few seconds worth of content) of a resource would be popular at any given time—those parts currently streamed. This window of activity advances continuously with the content playout point, while older packets will no longer see any requests, and will be dropped due to the per-resource LRU mechanism.

6. APPLICATION EXAMPLES

MRTCP-aware applications use an extended variant of the Berkeley Sockets API. When establishing a TCP connection, labeling is turned off by default. A sender can issue control commands—in our implementation this is currently a *send()* system call with the label contained in the buffer and a flag indicating that this call contains a label, although the protocol is not tied to the details of the API—to set or change a label, or to stop labeling.[4] Such a marking will is reflected in the local send buffer; all subsequent data will use the corresponding label until the next control command. The sender uses the same system calls for sending data as usual such as *write()*, *writev()*, *sendfile()*, etc.

To ensure that packets are easily cachable and reusable, the sender must ensure that content is always split at the same packet boundaries when fetched by different receivers, or using different offsets. To do this, the server application explicitly controls the initial boundary for each label relative to the content offset and the subsequent segment size to use both subject to MSS and MTU size limitations).

Content labels are removed by the receiving implementation before data is passed to the application, so that the application does not notice if it has received labeled data. Even if a receiver is MRTCP-aware, all handling is done inside the kernel.

The implementation of applications using these primitives is straightforward. We will discuss this in two examples: a web server and a live streaming server.

6.1 Web server

A web server opens a TCP socket and configures MRTCP support. It then begins accepting regular HTTP requests. For each request, the server takes the request URI, the client-side preferences (e.g., based upon the *Accept* and the *User-Agent* headers), and any cookies included in the request into account, and resolves the request to a version of a resource. It computes a label for the resource in a way that repeated requests for the same version of the resource will yield the same label. This may be achieved by using a cryptographic hash over the data comprising the content to be returned (and thus could be pre-computed). Finally, it initializes the content sequence number to the offset where retrieval shall start; typically this is zero, but different offsets can be requested in an HTTP *Range* header.

The server *write()*s the HTTP response header, including the CRLF separating it from the HTTP body, to the TCP socket. The server then sets the content label and the content sequence number, and writes the body of the HTTP response. After the end of the HTTP body, the server issues another control command to clear the label again. Finally, the server returns to processing the next request.

This simple procedure allows HTTP connections to be kept alive for retrieval multiple objects and is compatible with pipelining and the use of multiple parallel TCP connections. It works naturally for (adaptive) HTTP streaming of video content, as the latter is built on top of basic HTTP primitives; in this case, content encoded at different bit rates would simply use different labels.

The operations for retrieving chunks of data in peer-to-peer systems follow a similar scheme, except that they use different means for content identification, and may use bidirectional data transmission.

6.2 Synchronous (multicast-style) streaming

Synchronous media streaming, as often used for IPTV, differs from the above scenarios in one important respect: a receiver requesting a stream at time t_0 will receive the program feed starting from what was "aired" at that very moment, while another receiver joining at $t_1 > t_0$ will not receive the content sent prior to t_1 (with the possibly exception of some video I-frames from the past, needed for seamless codec operation). This effectively means that a request for a synchronous media stream can be seen as a request for a content item at a given initial offset, and that this offset is implied by the time at which the request is issued.

An MRTCP-capable IPTV server would treat each program channel as a content item of potentially infinite size with changing labels at different offsets. The sender only needs to maintain the presently used label and the count of bytes sent using this label, i.e., the content sequence number. Such a server resolves a request for a program channel as described for HTTP above, but also takes into account the current playback time when the request was received and translates this into the current label and offset for the stream content item (or the next/previous suitable entry point for a receiver). After sending the response header in the respective protocol (assume HTTP again, for example) it then simply configures the label and offset and then sends the response data as before. This works for plain as well as for adaptive HTTP streaming.

7. INITIAL EXPERIMENTATION

In this section, we briefly report on our proof-of-concept implementation. We use it to perform some a small set of real-world experiments to test interoperability with existing client implementations, and to assess the fundamental feasibility of our approach. (We discuss some of the deployment issues along with further perspectives in the next section.)

We have implemented the sender-side MRTCP algorithm in the Linux kernel[5]. The implementation requires a modest amount of modifications in sender-side TCP algorithms, to add the MRTCP_DATA options in packets, using a special flag for send call to allow applications pass the content label to the kernel, and processing of the incoming acknowledgments with MRTCP_ACK option.

To assess the suitability of MRTCP in the real world, we developed a prototype of an MRTCP implementation for a network node as a bump-in-the-wire, *mrtcp-bridge*, that performs L2 forwarding of frames between two Ethernet interfaces. The purpose is primarily to validate that an MRTCP router implementation can be done using the specification above and to demonstrate interoperability with unmodified receivers; we do not aim for performance measurements at this point.

mrtcp-bridge is written in C (some 1,400 LoC), runs on Linux, and uses packet sockets to capture and forward packets as well as to send cached replies. It implements the stateful version of an

[4]We use *send()* to be able to re-use the same code base with ns-3.

[5]We use Linux kernel version 2.6.26, which is the most recent version supported by the Network Simulation Cradle packaged with ns-3.9.

MRTCP router and as such tracks resources, flows, and packets, with a configurable number of entries for each. Hash tables are used for efficient matching. The data structures allow for the implementation of diverse purging policies, of which we presently use LRU for resources, packets, and flows.

As a server, we chose a lightweight open source web server, *lighttpd*[6] version 1.4.18 (for which the H.264 HTTP streaming extension[7] is readily available). For our initial experimentation, we assume static and browser-independent resources and compute use as a label the first 8 bytes of an MD5 hash over the request URI. As described in section 6, we set the label after writing the HTTP headers and reset it when the message body is finished. The chunk-based write queue of *lighttpd* makes it easy to add identifiable labels at the right positions into the data stream and initiate the corresponding control commands accordingly. Overall, less than 30 lines of code needed to be changed or added.

For simple interoperability experiments we connect the server through a separate machine running *mrtcp-bridge* to our lab network and issue web requests from different web browsers and *wget* to retrieve copies of the authors' home pages and documents stored on the server.

The simple experiments confirm that the use of TCP options does not impact interoperability across the operating systems we tried: MacOS Leopard and Snow Leopard, different versions of Debian and Ubuntu Linux, and Microsoft Windows XP and Windows 7 as well for mobile phones Android (HTC Nexus One), iPhone OS, Linux (Nokia N900), and Symbian S60 (Nokia E71).

We also find that, in all cases, the *mrtcp-bridge* can feed packets from its cache into different TCP connections, to the same and different hosts. As expected given the light load, the resources are completely stored on the *mrtcp-bridge* so that only the SYN/ACK, request/response header, and the FIN/ACK handshakes take place end-to-end—and the first labeled packet always comes from the server as this is needed to prime the router.

8. DISCUSSION

8.1 Content label namespace

MRTCP identifies content items using labels and uses a content sequence number for addressing segment-sized pieces of an item. Moreover, we assume that the label name space is managed by each server and therefore can include the server IP address and port number in the identification process (thus effectively enlarging the name space). This has the advantage that the label name space becomes implicitly hierarchically managed as the server process becomes responsible for its own space and can actively avoid clashes. The disadvantage is that the content becomes tied to a single server, even if multiple server could serve the same content independently, use coherent labeling, and ensure similar packetization boundaries.

Our present experimental implementation uses 8 bytes labels as a tradeoff between limited TCP option space and the probability of label collisions. As long as we use per-server labels, the available space should suffice to guarantee statistical uniqueness when using hash functions to compute the labels (and a server data base could recognize any clashes offline) or to allow enumerating all versions of the stored content items.

When wanting to share the identifiers across servers to better exploit redundancy, the label space will need to grow. Since coordination can no longer be reasonably achieved, using, e.g., 160-bit SHA-1 hashes (that still fit into TCP options) would be advisable. Such a label would be self-certifying. MRTCP-aware receivers can validate such self-certifying labeled content items. In this case, the items should just be reasonably short—similar to *chunks* in CCN or HTTP streaming—so that validation can take place before the content gets passed to the application and processed. However, legacy receivers cannot perform such validations and would suffer from clashes, making them vulnerable to accidental clashes or intentionally mislabeled content. This calls for explicit end-to-end content protection at the application layer (irrespective of the underlying transport) to detect false content—which should be pursued in an information-centric Internet anyway.

[6]http://www.lighttpd.net/

[7]http://h264.code-shop.com/trac/wiki

8.2 Cache pollution and DoS attacks

MRTCP routers do not track individual connections for caching packets but only look at labeled packets instead. This allows an attacker to send large numbers of labeled segments towards a peer to pollute caches en route with false data packets. The label and sequence increments to use could easily be learned by accessing the sender and retrieving the same resource. One protection against this in routers, besides the ingress filtering at the ISP level and RPF checks in the router, is accepting packets only from an active connection between a receiver and a sender that is in the flow table. A new flow could be activated only after a grace period in which ACKs were sent to the sender and no RST packets were seen in response. This would essentially rely on the protection mechanisms against third-party traffic injection inherent to TCP.

If injecting false packets does not succeed, an adversary could still try to launch a denial of service attack against a router by sending volumes of arbitrarily labeled packets some of which the router would store. The mechanism of prioritizing accepting incoming packets from existing over new flows helps here (and the attacker might have to send data at a high rate to achieve the desired effect). And even if this de-facto disturbs one router along a path, there are others available along the path to a server who can jump in.

Finally, adversaries might fake ACK packets to trigger data packet transmission to unsuspecting receivers. The flow state in stateful MRTCP routers will prevent this (as ACKs do not create flow state). Even though stateless MRTCP routers might reply if there is a matching packet, this would not support amplification attacks as the number of generated packets per ACK is limited.

8.3 Middleboxes

MRTCP uses TCP options for its signaling, relying on its options being passed end-to-end. Even though TCP server implementations ignore unknown options, connections may fail upon setup or mid-connection if TCP options included cause middleboxes to drop packets. Earlier experiments [19] hinted that TCP connection establishment only fails rarely (0.2%) because of unknown TCP options and also that the use of options in the middle of a connection only leads to some 3% of failures. Recent findings [8] showed that out of the top 10,000 web sides listed by Alexa only some 0.15% did not respond to unknown TCP options in the SYN packets, again likely due to the SYN packets being dropped by middleboxes. The authors are not aware of more recent detailed studies on the behavior of middleboxes with respect to TCP options, but the above results appear promising to achieve at least backward-compatible operation.

While the translation of IP addresses and port numbers in NAT devices is not an issue for MRTCP, some of those were also found to modify TCP sequence numbers [10, 19]. Since MRTCP only communicates the content sequence numbers in its options and implies the mapping from the TCP sequence number, the mappings will remain intact and MRTCP operate correctly across such mid-

dleboxes.

MRTCP expects the segment boundaries to remain unchanged when the packets traverse the network. As TCP does not provide any guarantees to that end, some middleboxes re-align packet boundaries as they see fit. It remains for further study, what the impact of unknown TCP options (that differ at least in the content sequence) on the behavior of such middleboxes would be. Making packets small or larger or gluing them together would destroy the proper mapping between the content sequence number and the TCP sequence number; moreover, a shift in offsets would make cache matches unlikely.

Finally, any application layer gateway (e.g., a proxy or cache) that terminates a TCP connection will appear as the remote endpoint to the sender.

9. OTHER RELATED WORK

MRTCP is designed to enable pervasive content caching, to support packet reuse across transport connections. Such reuse is intended to be useful for both synchronous and time-shifted content replication. The motivation behind MRTCP came from the data-oriented and content-centric networking architectures, such as DONA [16], PSIRP [15, 23], and CCN [13], and the thought that a useful, general purpose, content caching and distribution infrastructure could be designed to fit within the current Internet architecture.

To some extent, MRTCP could be viewed a probabilistic variant of *redundancy elimination* in which repeated transmission of the same data across (costly links inside) the network is avoided using by fingerprinting the contents of packets or data streams, caching contents, and replacing repeated content by small tokens that can be removed to restore the original data downstream of a bottleneck [20]. Many proposals have been made to this end. For example, *snoop* [6] suggested packet-level retransmissions to cope with losses on wireless links within the same TCP connection context, and numerous commercial "WAN optimizer" products implement these ideas. This architecture is, however, restricted to closely cooperating caches, e.g., at both ends of a capacity-constrained link. More recent proposal take redundancy elimination at the packet level further by extending the concept to network-wide operation within and/or across domains [3, 4] domains. These approaches, however, require coordination across nodes, updated routing protocols and broad support in routers.

Other systems focus on redundancy elimination specifically for synchronous content distribution. CacheCast [21] reduces the traffic volume for datagram traffic across a path segment by avoiding repeated transmission of the same payload across a link (within a short time window) and sending summary packets instead to cooperating routers. This takes a first towards packet content re-use across different unicast flows for simultaneous multi-destination delivery but it does not support reliable transport protocols, works best only for direct links, and targets short time windows only, limiting its applicability to caching.

Several proposals have been made to add multicast support to TCP. These include single connection emulation across a multicast group [22], and to use of multicast to augment unicast TCP [18, 14]. As with Scalable Application layer Multicast (SAM) [7], and the numerous peer-to-peer content delivery protocols, these typically rely on overlay nodes that are explicitly addressed when setting up connections, and hence require wholesale protocol upgrades to gain benefit.

Pull-patching [12] realizes a unicast-multicast combination at the application layer, combining multicast delivery of media streams with HTTP requests for patching the multicast stream where needed, e.g. for startup synchronization and for repair.

In addition, the ideas in MRTCP have been influenced by the work done in the mid-late 1990s on reliable multicast transport protocols (e.g., in the context of the IRTF reliable multicast research group[8] and later in the IETF reliable multicast transport working group[9]). Some proposals such *Pretty Good Multicast (PGM)* [9] allow for router support for reliable data delivery within a multicast transport group. All TCP enhancements and reliable multicast-related approaches share that they only support synchronous multi-destination delivery and require a common group to be explicitly set up. Moreover, most of them require wide-area multicast support for efficient operation as well as client side modifications.

10. CONCLUSIONS

Content-centric applications generate the majority of the traffic in the Internet today. They are supported by a mixture of ad-hoc and application-specific engineering solutions, but there is no unified and application-neutral framework for pervasive content caching and dissemination.

We have presented MRTCP, a TCP enhancement that can provide flow-independent in-network content caching in both synchronous pseudo-multicast and time-shifted modes, while maintaining sender visibility into downloads. By adding content labels to data packets, and making server side TCP modifications to maintain data framing, we allow MRTCP routers to effectively cache data. To work with legacy (unmodified) TCP receivers, routers maintain a per-flow mapping from TCP state to content identifiers. A stateless variant of MRTCP is also presented, for scenarios where clients can be upgraded.

Future work will consider in-network loss repair, rather than end-to-end retransmission. This can improve performance, but must be implemented in a manner that provides feedback to the sender about loss, for quality of experience monitoring. The trade-off between in-network state and ability to perform sophisticated congestion control between middlebox and receiver will also be considered further.

MRTCP provides a poor man's approach to content-centric networking. While we do not provide all the benefits of a clean-slate approach, by operating within the existing architecture, we gain incremental and mixed deployment, and the ability to effectively cache named content within the network in an application-neutral, location-transparent, manner.

Acknowledgement

The authors would like to thank Pekka Nikander for discussions and useful suggestions during the early part of this work. The research in this paper has been partly funded by the Cisco University Research Program and by the EC FP7 PURSUIT project under contract ICT-2010-257217.

11. REFERENCES

[1] S. Akhshabi, A. C. Begen, and C. Dovrolis. An Experimental Evaluation of Rate Adaptation Algorithms in Adaptive Streaming over HTTP. In *Proc. of ACM MM Systems*, 2011.

[2] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681, September 2009.

[3] A. Anand et al. Packet caches on routers: the implications of universal redundant traffic elimination. In *Proc. SIGCOMM*, 2008.

[8] http://rmrg.east.isi.edu/

[9] http://datatracker.ietf.org/wg/rmt/charter/

[4] A. Anand, V. Sekar, and A. Akella. SmartRE: an architecture for coordinated network-wide redundancy elimination. In *Proc. SIGCOMM*.

[5] S. Arianfar, P. Nikander, and J. Ott. On Content-Centric Router Design and Implications. In *Proc. of the ACM CoNext ReArch workshop*, 2010.

[6] H. Balakrishan, S. Seshan, E. Amir, and R. H. Katz. Improving TCP/IP Performance over Wireless Networks. In *Proc. Mobicom*, November 1995.

[7] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *Proc. SIGCOMM*, Pittsburgh, PA, USA, August 2002.

[8] Andrea Bittau, Michael Hamburg, Mark Handley, David Mazieres, and Dan Boneh. The case for ubiquitous transport-level encryption. In *Proceedings of USENIX Security*, 2010.

[9] T. Speakman et al. PGM Reliable Transport Protocol Specification. Experimental RFC3208, 2001.

[10] S. Gupta and P. Francis. Characterization and Measurement of TCP Traversal through NATs and Firewalls. In *ACM IMC*, 2005.

[11] Hung-Yun Hsieh, Kyu-Han Kim, Yujie Zhu, and Raghupathy Sivakumar. A receiver-centric transport protocol for mobile hosts with heterogeneous wireless interfaces. In *Proceedings of ACM MOBICOM '03*, San Diego, CA, USA, September 2003.

[12] E. Jacobson, C. Griwodz, and P. Halvorsen. Pull-Patching: A Combination of Multicast and Adaptive Segmented HTTP Streaming. In *ACM Multimedia*, 2010.

[13] V. Jacobson et al. Networking Named Content. In *Proc. CoNEXT*, December 2009.

[14] K. Jeacle et al. Hybrid Reliable Multicast with TCP-XM. In *Proc. CoNEXT*, 2005.

[15] P. Jokela et al. LIPSIN: Line Speed Publish/Subscribe Inter-Networking. In *Proc. SIGCOMM*, 2009.

[16] T. Koponen et al. A Data-Oriented (and Beyond) Network Architecture. In *Proc. SIGCOMM*, Kyoto, Japan, August 2007.

[17] C. Labovitz et al. Internet inter-domain traffic. In *Proc. SIGCOMM*, 2010.

[18] S. Liang and D. Cheriton. TCP-SMO: Extending TCP to Support Medium-Scale Multicast Applications. In *Proc. of IEEE INFOCOM*, 2002.

[19] A. Medina, M. Allman, and S. Floyd. Measuring the Evolution of Transport Protocols in the Internet. *ACM SIGCOMM CCR*, 35(2):37–52, April 2005.

[20] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proc. SIGCOMM*, 2000.

[21] P. Srebrny, T. Plagemann, V. Goebel, and A. Mauthe. CacheCast: Eliminating Redundant Link Traffic for Single Source Multiple Destination Transfers. In *Proc. Intl. Conf. Distributed Computing Systems*, 2010.

[22] R. Talpade and M. H. Ammar. Single Connection Emulation: An Architecture for Providing a Realible Multicast Transport Service. In *Proc. of the IEEE Internation Conference on Distributed Computing Systems*, 1995.

[23] D. Trossen, M. Särelä, and K. Sollins. Arguments for an information-centric internetworking architecture. *SIGCOMM CCR*, 40(2):26–33, 2010.

[24] H. Xie, Y. R. Yang, A. Krishnamurthy, Y. Liu, and A. Silberschatz. P4P: Provider portal for applications. In *Proc. SIGCOMM*, Seattle, WA, USA, August 2008.

Aalto-ST 5/2011

ISBN: 978-952-60-4068-4 (pdf)
ISSN-L: 1799-4896
ISSN: 1799-490X (pdf)

Aalto University
School of Electrical Engineering
Department of Communications and Networking
aalto.fi

BUSINESS + ECONOMY

ART + DESIGN + ARCHITECTURE

SCIENCE + TECHNOLOGY

CROSSOVER

DOCTORAL DISSERTATIONS